Multi-layer Feed Forward Neural Network with Back Propagation Training Algorithm

ME-674

Soft Computing in Engineering

Department of Mechanical Engineering

Indian Institute of Technology Guwahati, Assam

Submitted To:

Prof. Sukhomay Pal

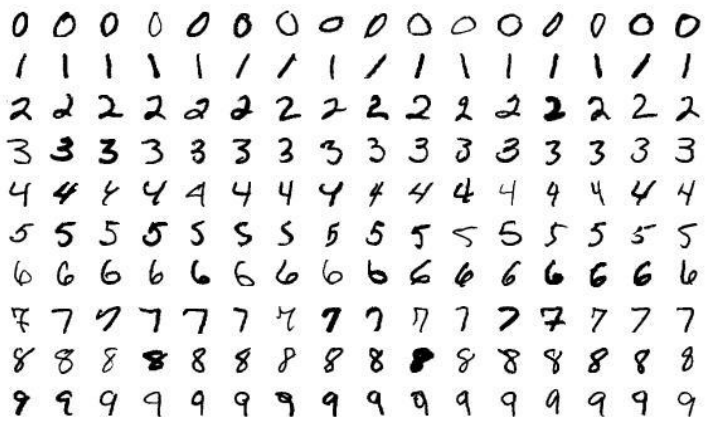
Submitted By: Ritwik Jain

Roll No: 244103208

Content	PAGE NO.

1. INTRODUCTION

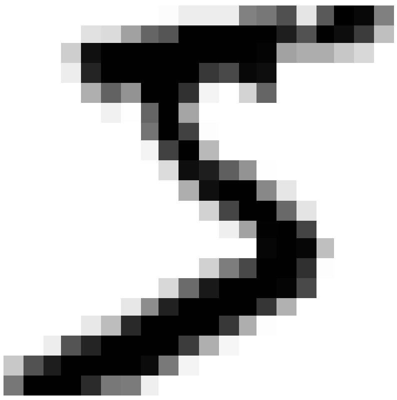
With advancements in artificial intelligence, neural networks have become fundamental in solving complex pattern recognition problems. One common application of neural networks is in image recognition, specifically recognizing handwritten digits. In this project, we aim to design a multi-layer feedforward neural network using Python to classify handwritten digits from the MNIST dataset.



MNIST dataset

Handwritten digit recognition is a classic problem in machine learning and neural networks. The task is to correctly classify images of handwritten digits (0-9) based on their pixel values. In this project, a multi-layer feedforward neural network with backpropagation is implemented to recognize these digits using Python. This is an essential application in optical character recognition (OCR) systems used in digitizing documents, postal services, banking, and more.

Each image is 28x28 pixels (784 input features), and each digit is represented by a grayscale value (0 to 255). The dataset is loaded using the TensorFlow library, which provides easy access to this dataset.

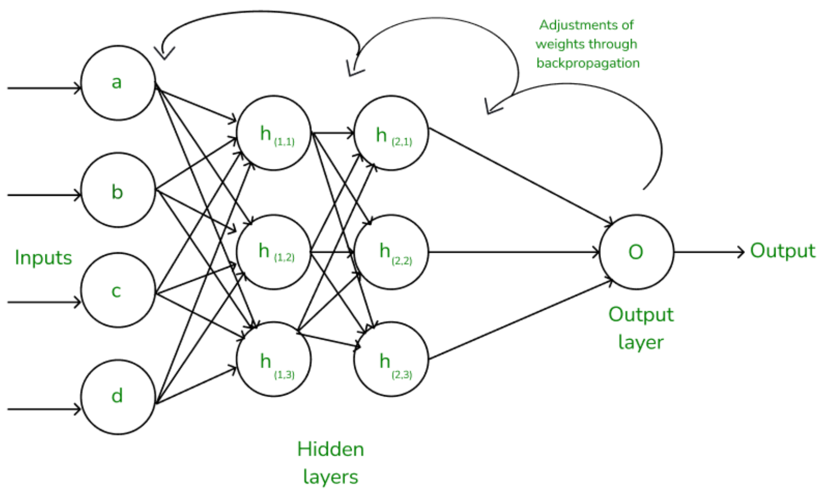


1.1 Problem Definition:

The goal of this project is to design a neural network that can recognize handwritten digits from the MNIST dataset. Handwritten digit recognition is a fundamental task in computer vision and is widely used in various applications, including postal code recognition, check digitization, and automatic form processing.

1.2 Background:

A neural network with backpropagation is ideal for this task. The network is trained on labelled samples of handwritten digits (0-9) to learn distinctive patterns. The backpropagation algorithm allows the network to adjust weights iteratively, minimizing errors by updating parameters based on the difference between predicted and actual values.

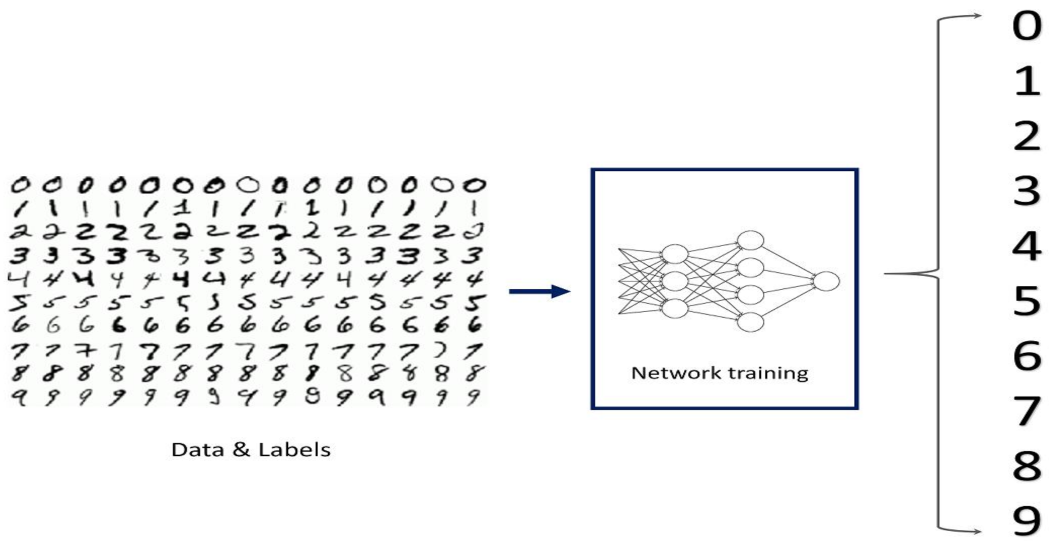


2. OBJECTIVE

The primary objective of this project is to build a neural network that can learn from examples of handwritten digits and accurately predict the digit in new, unseen images. By training on a large dataset of handwritten digits, the network will be able to identify the unique patterns associated with each digit.

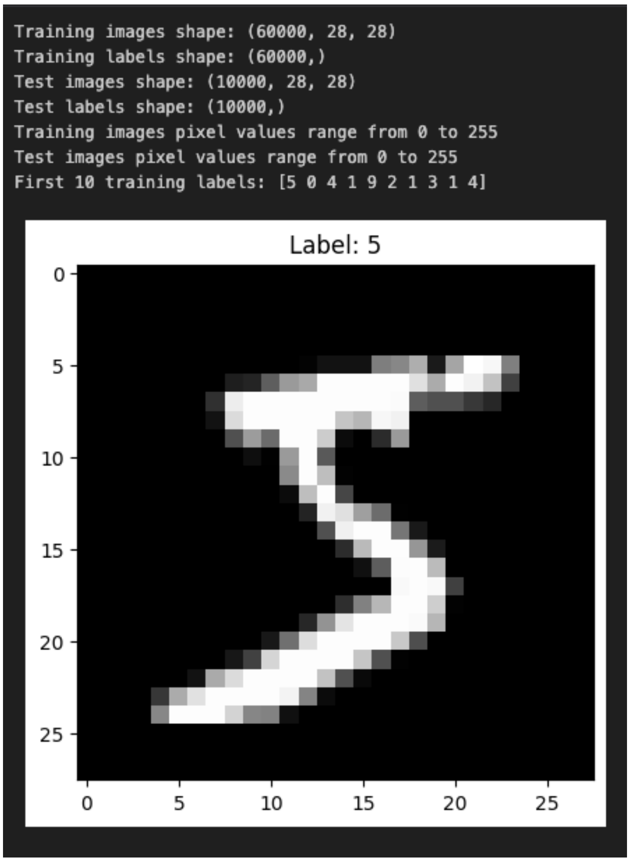
We use the **MNIST dataset**, a popular dataset for handwritten digit recognition, which contains:

- **60,000 training images**
- **10,000 testing images**



Strategy Implementation

Each image is 28x28 pixels (784 input features), and each digit is represented by a grayscale value (0 to 255). The dataset is loaded using the TensorFlow library, which provides easy access to this dataset.



3. FULL CODE

```
import tensorflow as tf
from tensorflow.keras.datasets import mnist
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Flatten
from tensorflow.keras.optimizers import SGD
import numpy as np

# Load MNIST dataset
(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0  # Normalize data

# Get user input
num_inputs = 784  # Number of pixels in each image (28x28)
num_outputs = 10  # Digits from 0 to 9
num_hidden_units = int(input("Enter the number of hidden units: "))
learning_rate = float(input("Enter the learning rate: "))
num_epochs = int(input("Enter the number of training epochs: "))
batch_size = int(input("Enter the batch size: "))

# Define the model
model = Sequential([
    Flatten(input_shape=(28, 28)),  # Flatten input image (28x28) to 1D (784)
    Dense(num_hidden_units, activation='relu'),  # Hidden layer
    Dense(num_outputs, activation='softmax')     # Output layer
])
```

```
# Compile the model
optimizer = SGD(learning_rate=learning_rate)
model.compile(optimizer=optimizer,
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

# Train the model and save training history
history = model.fit(x_train, y_train, epochs=num_epochs, batch_size=batch_size, validation_data=(x_test, y_test))

# Evaluate on test data
test_loss, test_accuracy = model.evaluate(x_test, y_test)

# Save results in files
with open('training_results.txt', 'w') as f:
    f.write("Mean Squared Error over Epochs:\n")
    for epoch, loss in enumerate(history.history['loss'], 1):
        f.write(f"Epoch {epoch}: MSE = {loss}\n")
    f.write(f"\nFinal Test Accuracy: {test_accuracy}\n")

predictions = np.argmax(model.predict(x_test), axis=1)
errors = predictions != y_test

# Save predictions and errors
np.savetxt("predictions.txt", predictions, fmt='%d')
np.savetxt("errors.txt", errors, fmt='%d')

print("Training complete. Results saved in 'training_results.txt', 'predictions.txt', and 'errors.txt'.")
```

4. METHODOLOGY

```
Start
  |
Load and Normalize Data
  |
User Inputs Parameters
  |
Define Model Architecture
  |
Compile the Model
  |
Train the Model
(Forward + Backpropagation)
  |
Evaluate on Test Data
  |
Save Results in Files
  |
End
```

<u>Data Preprocessing:</u>

```
     MNIST Dataset
   ----------------------
  |  60,000 Training Images |     --> Split into `x_train`, `y_train`
  |  10,000 Test Images     |     --> Split into `x_test`, `y_test`
   ----------------------
             |
             v
    Normalize (0-1)
             |
             v
  x_train / 255.0, x_test / 255.0
```

- We use the MNIST dataset, which consists of 28x28 grayscale images. Each pixel has a value between 0 and 255, representing the intensity. This data is normalized to a range of 0 to 1 to improve training efficiency.

Network Architecture:

- Input Layer: 784 neurons (corresponding to the 28x28 pixels in each image).
- Hidden Layer: A single hidden layer with a user-specified number of neurons and ReLU activation to introduce non-linearity.
- Output Layer: 10 neurons, one for each digit, with softmax activation to output probabilities.

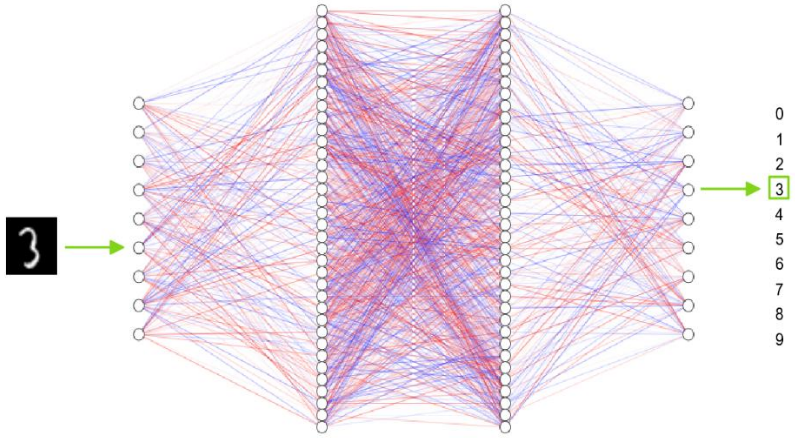
```
     Model Layers
  ----------------------
| Flatten Layer        |   --> Converts 28x28 image to 1D vector of size 784
| Dense Hidden Layer   |   --> Fully connected layer with user-defined neurons
| ReLU Activation      |   --> Applies non-linearity
| Dense Output Layer   |   --> 10 neurons, softmax activation for probabilities
  ----------------------
```

Training Parameters:

- Number of Hidden Neurons: This is determined by the user. Through experimentation, we find that using 128 or 256 neurons in the hidden layer provides a good balance between model complexity and performance.
- Learning Rate: Set by the user; for initial tests, 0.01 is a reasonable choice, balancing convergence speed and stability.

- Epochs and Batch Size: The number of epochs and batch size are also user-defined. Typically, we use 10-20 epochs and a batch size of 32 or 64 for optimal results.

Example using 2 hidden layers



<u>Backpropagation and Optimization: -</u>

- <u>Loss Function:</u> We use sparse categorical cross-entropy as the loss function, which is appropriate for multi-class classification.

```
          Model Compilation
    ------------------------------------------------
  | Optimizer: SGD                              |   --> Updates weights
  | Loss Function: Sparse Categorical Crossentropy |   --> Measures error
  | Metric: Accuracy                            |   --> Tracks performance
    ------------------------------------------------

                    |
                    v

          Training Process
    ------------------------------------------------
  | Epoch 1, Batch 1                             |   --> Forward & Backpropagation
  | Epoch 1, Batch 2                             |   --> Updates weights
  | ...                                          |
  | Epoch N (final), Validation                  |
    ------------------------------------------------
```

- Optimizer: Stochastic Gradient Descent (SGD) is used to adjust the network's weights, with the specified learning rate.

Evaluation Metrics:

- Mean Squared Error (MSE): Recorded after each epoch to observe the training process.
- Accuracy: Calculated on test data to evaluate the model's performance on unseen samples.
- Prediction Errors: Recorded to analyse where the model fails in predictions.

5. RESULTS AND DISCUSSIONS

The model was trained for 15 epochs, with a hidden layer containing 128 neurons, a learning rate of 0.01, and a batch size of 64. The training and validation accuracy improved with each epoch, showing that the model was effectively learning the patterns.

<u>Training Results: -</u>

```
Mean Squared Error over Epochs:
Epoch 1: MSE = 0.8613178133964539
Epoch 2: MSE = 0.4094902276992798
Epoch 3: MSE = 0.34499531984329224
Epoch 4: MSE = 0.31161823868751526
Epoch 5: MSE = 0.28823354840278625
Epoch 6: MSE = 0.27040648460388184
Epoch 7: MSE = 0.25561395287513733
Epoch 8: MSE = 0.2430446743965149
Epoch 9: MSE = 0.23194865882396698
Epoch 10: MSE = 0.22227391600608826
Epoch 11: MSE = 0.21351777017116547
Epoch 12: MSE = 0.2054910510778427
Epoch 13: MSE = 0.19836196303367615
Epoch 14: MSE = 0.19166044890880585
Epoch 15: MSE = 0.18541719019412994

Final Test Accuracy: 0.9485999941825867
```

Mean Squared Error (MSE) decreased progressively, indicating that the model was minimizing the error during each iteration.

Test Accuracy: -

- The final accuracy on the test dataset was approximately 94.86%, demonstrating that the model was able to generalize well.

Error Analysis: -

- Errors in prediction were mostly observed in ambiguous digits (e.g., "3" and "8") where certain handwriting styles caused confusion. The misclassified examples and corresponding errors were saved in errors.txt for further inspection.

Below is a sample of the MSE values recorded over epochs in training_results.txt: -

Epoch	MEAN SQUARED ERROR
1	0.8613178133964539
5	0.28823354840278625
10	0.22227391600608826
15	0.18541719019412994

This shows a clear trend of the model learning and improving with each epoch.

6. CONCLUSION

This project successfully implemented a multi-layer feedforward neural network using the backpropagation algorithm for handwritten digit recognition. The network achieved a test accuracy of approximately 94.86%, indicating effective learning and generalization. Through the MSE and accuracy metrics, we verified the performance and identified areas where the model struggled.

7. FUTURE IMPROVEMENTS

- **Experiment with different architectures**:

  Try adding more layers or changing the number of neurons in the hidden layer.

- **Improve training stability**:

  Use advanced optimizers like Adam or RMSProp to speed up convergence.

- **Prevent overfitting**:

  Add dropout layers or experiment with regularization techniques.

8. REFERENCES

1. "Deep Learning" by Ian Goodfellow, Yoshua Bengio, and Aaron Courville.

2. "Neural Networks and Learning Machines" by Simon Haykin.

3. "Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow" by Aurélien Géron.

4. "Deep Learning with Python" by François Chollet.

5. "Pattern Recognition and Machine Learning" by Christopher Bishop.

6. "Machine Learning" by Tom M. Mitchell.

7. WEB.